Acquisitions

From: icase-cs-techreports-owner[SMTP:icase-cs-techreports-owner@icase.edu]
Sent: Monday, September 08, 1997 9:24 AM
To: icase-cs-techreports; icase-all-pubs; shelly
Subject: ICASE Reports No. 97-29 and 97-37 now on-line

ICASE Report No. 97-29
NASA CR-201716

BINARY DISSECTION: VARIANTS & APPLICATIONS

Shahid H. Bokhari, Thomas W. Crockett, and David M. Nicol

Partitioning is an important issue in a variety of applications. Two examples are domain decomposition for parallel computing and color image quantization. In the former we need to partition a computational task
over many processors; in the latter we need to partition a high resolution color space into a small number of representative colors. In both cases, partitioning must be done in a manner that yields good results as defined by an application-specific metric.

Binary dissection is a technique that has been widely used to partition non-uniform domains over parallel computers. It proceeds by recursively partitioning the given domain into two parts, such that each part has approximately equal computational load. The basic dissection algorithm does not consider the perimeter, surface area or aspect ratio of the two
sub-regions generated at each step and can thus yield decompositions that have poor communication to computation ratios.

We have developed and implemented several variants of the binary dissection approach that attempt to remedy this limitation, are faster than the basic algorithm, can be applied to a variety of problems, and are amenable to parallelization.

The Parametric Binary Dissection (PBD) algorithm tries to minimize the difference between {\em volume} $+$ $\lambda \times (${\em surface}$)$ for each of the two subregions it generates at each step. When applied to parallel computing, {\em volume} represents the amount of computation required while {\em surface} is proportional to interprocessor
communication. The parameter λ permits us to trade off load imbalance against communication overhead. When λ is zero, the algorithm reduces to simple binary dissection.

The Fast Adaptive Dissection (FAD) algorithm is used for color image quantization, where samples in a high resolution color space are mapped into a lower resolution space in a way that minimizes color error. In
this case the algorithm tries to minimize the product {\em popularity} $\times$ {\em color error} for the resulting sub-regions. {\em popularity} is the number of colors in a region while {\em color error} is the maximum distance between points within a region.

We describe the performance of PBD and FAD on a variety of representative problems and present ways of parallelizing the PBD algorithm on 2- or 3-d meshes and on hypercubes.

Binary Dissection: Variants & Applications

Shahid H. Bokhari
Department of Electrical Engineering
University of Engineering & Technology
Lahore, Pakistan

1997

Thomas W. Crockett
ICASE
NASA Langley Research Center
Hampton, Virginia

David M. Nicol
Department of Computer Science
Dartmouth College
Hanover, New Hampshire

Abstract

Partitioning is an important issue in a variety of applications. Two examples are domain decomposition for parallel computing and color image quantization. In the former we need to partition a computational task over many processors; in the latter we need to partition a high resolution color space into a small number of representative colors. In both cases, partitioning must be done in a manner that yields good results as defined by an application-specific metric.

Binary dissection is a technique that has been widely used to partition non-uniform domains over parallel computers. It proceeds by recursively partitioning the given domain into two parts, such that each part has approximately equal computational load. The basic dissection algorithm does not consider the perimeter, surface area or aspect ratio of the two sub-regions generated at each step and can thus yield decompositions that have poor communication to computation ratios.

We have developed and implemented several variants of the binary dissection approach that attempt to remedy this limitation, are faster than the basic algorithm, can be applied to a variety of problems, and are amenable to parallelization.

The Parametric Binary Dissection (PBD) algorithm tries to minimize the difference between *volume* + $\lambda \times$ (*surface*) for each of the two subregions it generates at each step. When applied to parallel computing, *volume* represents the amount of computation required while *surface* is proportional to interprocessor communication. The parameter λ permits us to trade off load imbalance against communication overhead. When λ is zero, the algorithm reduces to simple binary dissection.

The Fast Adaptive Dissection (FAD) algorithm is used for color image quantization, where samples in a high resolution color space are mapped into a lower resolution space in a way that minimizes color error. In this case the algorithm tries to minimize the product *popularity* $\times$ *color error* for the resulting sub-regions. *popularity* is the number of colors in a region while *color error* is the maximum distance between points within a region.

We describe the performance of PBD and FAD on a variety of representative problems and present ways of parallelizing the PBD algorithm on 2- or 3-d meshes and on hypercubes.

This research was supported by the National Aeronautics and Space Administration under NASA Contract NAS-1-19480 while the authors were resident at the Institute for Computer Applications in Science & Engineering, NASA Langley Research Center, Hampton, Virginia. Shahid Bokhari was supported in addition by a grant from the Directorate of Research Extension and Advisory Services, University of Engineering & Technology, Lahore. David Nicol was supported in addition by NSF grant CCR-9201195.

Authors' email addresses: shahid@icase.edu, tom@icase.edu, nicol@cs.dartmouth.edu.

Production note: Due to limitations in the reproduction process, hardcopy versions of this report may not adequately convey the subtle variations in image quality in Figure 14. For best results, we recommend obtaining the digital version of this report (`ftp://ftp.icase.edu/pub/techreports/97/97-29.pdf`), and either viewing the images on a high-resolution 24-bit color monitor, or printing them on a photographic-quality output device.

1 Introduction

The partitioning of problems over the processors of a parallel computer system remains the subject of considerable research. This problem is particularly difficult when the domain or region being partitioned has nonuniform computational requirements. For example, in a climate model, some areas of the earth's surface may require greater computational effort than others. We would like to apportion parts of the problem domain over the processors of the system in such a way as to put equal computational load on all processors, thereby minimizing the total computational time. Another example is the solution of aerodynamic problems using "unstructured" meshes which are graphs embedded in 2- or 3-dimensional space [3]. Computation is carried out on the nodes of the graph, alternating with communication over the edges.

Problems of this type require huge amounts of computational power and are at the limits of the memory capacities of the largest parallel processors. There is a pressing need for techniques to improve the running time of such problems, because they require scarce and expensive resources for their solution and also because the solution itself has great economic value.

The binary dissection or orthogonal recursive partition algorithm was developed by Berger & Bokhari [1] as a means for partitioning non-uniform domains. This approach permits a very fast solution to the partitioning problem and has been successfully applied in practice [4]. This algorithm makes a series of bisections, along orthogonal directions, minimizing the load imbalance at each step. Since the partitioning attempts only to balance the load, the subregions generated may have poor communication to computation ratios. This is because the simple partitioning criterion does not consider the perimeter, surface area, or aspect ratio of the subregions being generated.

In the current paper we present two variants of the binary dissection algorithm. These are *Parametric Binary Dissection* (PBD) and *Fast Adaptive Dissection* (FAD).

In Parametric Binary Dissection each recursive cut is chosen to minimize $volume + \lambda \times (surface)$. When the domain is a 2-dimensional region, *surface* could refer to the perimeter of a subregion. In the 3-d case it could refer to the surface area. When PBD is used to partition embedded graphs, *volume* refers to the number of vertices in a region and *surface* the number of edges leaving a region. In general, *surface* is a measure of the communication overhead and the parameter λ permits us to trade off load imbalance against communication overhead. We can sacrifice some amount of load balance for better communication balance in order to obtain faster overall computation time. When λ is zero, the new algorithm reduces to simple binary dissection.

Fast Adaptive Dissection finds application in *color image quantization*, in which samples in a high-resolution color space are mapped onto a lower resolution space in a way that minimizes the color error. In our formulation of this problem, one is given a 3-dimensional grid in which some points are occupied and others are vacant. The objective is to partition this grid into regions such that (1) the total number of regions is bounded by some given maximum, and (2) a single value can be computed for each region which is a satisfactory representative of the enclosed points subject to some global error metric. We have found that a partitioning process which attempts to minimize the product *popularity* × *color error* yields good results at high speed. Here *popularity* is the number of colors in a region, while *color error* is the maximum distance between two points in the region.

Mesh partitioning is one of the problems to which we apply our new algorithms. A number of other partitioning strategies have been proposed for this problem, and it is worthwhile to compare our approach with existing work. The previous work, e.g. [8], is built around the notion of graph separators. In such a formulation a mesh is viewed as an undirected graph. An edge-separator is a set of edges that disconnects the graph into two nearly equal sized pieces. The goal of separator based approaches is to find separators of small size, thereby reducing the communication overhead. There are two principal differences between parametric binary dissection, and separator-based algorithms. PBD/FAD constrain all cuts to be straight lines, a constraint not imposed on the other methods. As a consequence, for certain problems and ranges of parameter values, the partitions produced by PBD/FAD on this application are almost certainly inferior. This deficiency is balanced by the fact that

- PBD/FAD are more general in their application (e.g. we see no easy way to use graph separators for the color quantization problem),
- linear cut constraints arise naturally in a number of applications, and
- PBD/FAD are undoubtedly the simplest, and likely fastest, methods among the alternatives.

Thus the quality of partitions produced by PBD on the specific problem of mesh partitioning is not the sole measure of its value. Furthermore, PBD is trivial to extend to graphs with weighted nodes and edges.

2 Binary Dissection

The original binary dissection algorithm can be applied to a variety of situations. In the present paper we are concerned with the partitioning of 2, 3 (or possibly higher) dimensional domains containing n points specified by their $x, y, z, \ldots$ coordinates. These points are bisected along the x direction by sorting the x coordinates and finding the mid-point. This process is accomplished in $O(n \log n)$ time for sorting and $O(n)$ time for splitting the list of points. The bisection process is then repeated along the y direction for the two subdomains and so on. If the *depth of partitioning* (the number of times the bisection is carried out) is given by d, then the entire process takes time

$$O(\sum_{i=0}^{d.\ 1}(2^i(n/2^i \log n/2^i) + n)) = O(dn \log n). \tag{1}$$

Since the depth of partitioning $d \leq \log n$, this results in $O(n \log^2 n)$ in the case of problems where the partitioning is carried out to large depths. However, in many problems of interest the depth of partition d is small compared to $\log n$ and it is more meaningful to use expression (1).

The basic bisection step described above can also be carried out using a fast ($O(n)$) median finding algorithm [2]. This eliminates sorting and we are left with $O(dn)$ time. However the constants involved in the median finding algorithm are large and this method remains of theoretical interest only.

Binary dissection partitions only on the basis of numbers of points and ignores their spatial distribution. As a result it can yield partitions that have poor aspect ratio (the ratio of largest to smallest sides), which may be undesirable in specific applications. When binary dissection is applied to the partitioning of graphs embedded in 2 or 3 dimensional space, as is the case in many important aerodynamic problems, the edge information (which determines the amount of information that needs to be communicated between points) is ignored. Thus while binary dissection can be (and has been) applied to such problems, the partitions obtained can sometimes be poor as far as the compute/communicate ratio is concerned.

3 Parametric Dissection

Parametric binary dissection remedies one of the shortcomings of the basic algorithm by explicitly taking surfaces of regions into account. Thus, if the problem is to partition a three dimensional region that contains a number of points, we minimize at each bisection step $volume + \lambda \times (surface)$ for the two subregions.

By *volume* we mean the number of points in each region—this is the quantity that simple binary dissection minimizes. *Surface* can refer to a variety of region properties. For example, if the problem is to partition a 2-dimensional region into subregions such that the resulting subregions are as square as possible, we may wish to use the perimeters of the resulting rectangles as our surface property. At each bisection step we would minimize the number of points in each rectangle plus λ times their perimeters. The parameter λ **permits us to trade off volume against surface—by sacrificing some amount of volume balance, we can improve the surface balance.**

The preceding example can be extended in an obvious fashion to 3 or higher dimensions. Various surface properties can be used. In the following discussion we shall assume that the surface property **can be computed easily, so that the analysis of time complexity of the algorithm is not affected** by it. We can use a complicated surface property if we are willing to pay for the time required to compute it while carrying out binary dissection.

The discussion so far has been in terms of point problems, where we are given a collection of points in 2, 3 or higher dimensional space. A more complicated situation arises when we are given a graph embedded in 2, 3 or higher dimensions. Each point or node has associated with it a set of coordinates and an adjacency list. The objective here is straightforward: each bisection minimizes $nodes + \lambda \times (edges\ cut)$.

Graph partitioning problems arise in many environments, most notably in the analysis of unstructured meshes. When such meshes are partitioned and mapped onto parallel computers, the running time is modeled by

$$\max_{all\ regions} [nodes\ in\ region + \lambda \times (edges\ leaving\ region)]. \tag{2}$$

Here λ corresponds to the *communicate to compute ratio* for the given parallel computer system, i.e. the ratio of time required to fetch a datum from a remote processor to the time to compute on a datum on the local processor. The time given by (2) is normalized to the time required to compute on one point, assuming a uniform computation cost for each point. The density of edges in unstructured meshes can vary enormously from one part of the domain to another and it is easy to see that taking edges into account can result in better partitionings. This would not be the case for embedded graphs in which the edges are more or less uniformly distributed. We are assuming that the communication overhead is proportional to the amount of data transmitted. This holds true for modern parallel machines with high performance communication hardware.

Figure 1 gives a simple example of how parametric dissection can lead to better partitions when

applied to graphs.

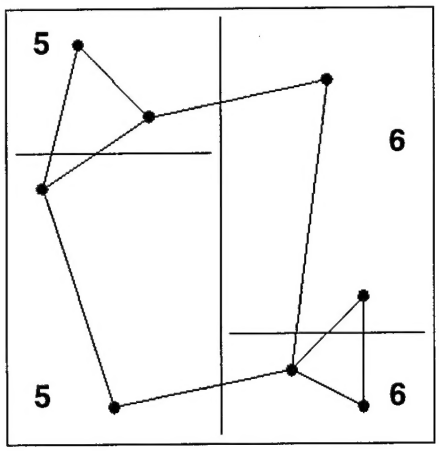
For the case of point problems the complexity of parametric binary dissection is unchanged at $O(dn \log n)$, where d is the depth of partitioning. For graph problems, we have to look at all edges before splitting, at every depth of the partition. Hence, the complexity becomes $O(\max[dn \log n, de])$, where e is the number of edges in the graph.

4 Fast Parametric Dissection

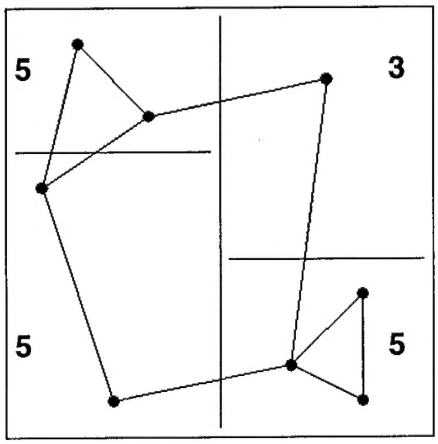
A major factor contributing to the time complexity of the binary dissection algorithms presented in Sections 2 and 3 is repeated sorting at each depth. We now show how parametric binary dissection can be accomplished by sorting only once per dimension by using a well known technique [12]. The fast algorithm we present also improves the time required for plain binary dissection.

A separate index list is created for each dimension. Element `i` of this list gives the index of the of the data point that is ith in sorted order. When a region is partitioned, all indices are split, so that the sublists corresponding to each subregion remain sorted. For purposes of exposition, we assume a 3-d graph partitioning problem and partition on the basis of expression (2) of Section 3.

Assume that the index lists for the x, y and z dimensions are stored in arrays `xlist[]`, `ylist[]` and `zlist[]`. The subregion to be partitioned is stored in array positions `L...U`. This means that the x dimension index list extends from `xlist[L]` to `xlist[U]` and so on. The current depth of partitioning is `depth`. This variable is initialized to be the maximum depth to which partitioning is to be carried out and is decremented at each level. The coordinates of point `i` are stored in `x[i]`, `y[i]`, `z[i]`. The parametric bisection is computed as follows.



(a)



(b)

Figure 1: **The benefit of parametric dissection. In part (a) plain binary dissection is applied to give perfect** node balance. The numbers in the four regions give the sum of nodes in region and edges leaving region (i.e., in the expression for time to execute (2) in main text, λ is assumed to be one). The maximum sum is 6. In part (b) parametric dissection is employed and results in maximum sum 5, as the new algorithm avoids cutting through regions with high edge density.

procedure PARAMETRIC CUT(depth,L,U,x,y,z,xlist,ylist,zlist);

1. Sweep forward from i=L to U counting the edges that would leave the left hand region, if the left hand region was L to i (inclusive). Store the result in leftvec[i].

2. Sweep backwards from i=U down to L counting the edges that would leave the right hand region, if the right hand region was i to U (inclusive). Store the result in rightvec[i].

3. Sweep forward again from i=L to U **to find the optimal split point:**

   - the left hand region comprises L to i,
   - the right hand region comprises i+1 to U
   - the optimal split point SPLITPLACE is the value of i for which the objective MAX((i −L+1) + λ×(leftvec[i]), (U−i) + λ×(rightvec[i+ 1])) is minimum. The value x[SPLITPLACE] is SPLITVALUE.

4. xlist has now been split into two parts, L to SPLITPLACE and SPLITPLACE+1 to U. The x coordinates of these points are already sorted since the original sorted index list is undisturbed.

5. Split the ylist: sweep forward from i=L to U moving successive values of ylist[i] for which x[ylist[i]] ≤SPLITVALUE **to the first part of the list (Figure 3 illustrates this for a 2-d** problem). The remaining values are moved to the second part of the list.

6. Similarly split the zlist.

7. All three indices xlist, ylist and zlist have now been split so that elements [L..SPLITPLACE] of these lists contain the points in one of the subregions and [SPLITPLACE+1..U] those in the other. When accessed through these lists the x, y and z coordinates of points are in sorted order.

8. Recursively cut for next depth but along next dimension:
   if(depth>1) then{
   PARAMETRIC_CUT(depth−1,L,SPLITPLACE,y,z,x,ylist,zlist,xlist)
   PARAMETRIC_CUT(depth−1,SPLITPLACE+1,U,y,z,x,ylist,zlist,xlist)}

end PARAMETRIC_CUT;

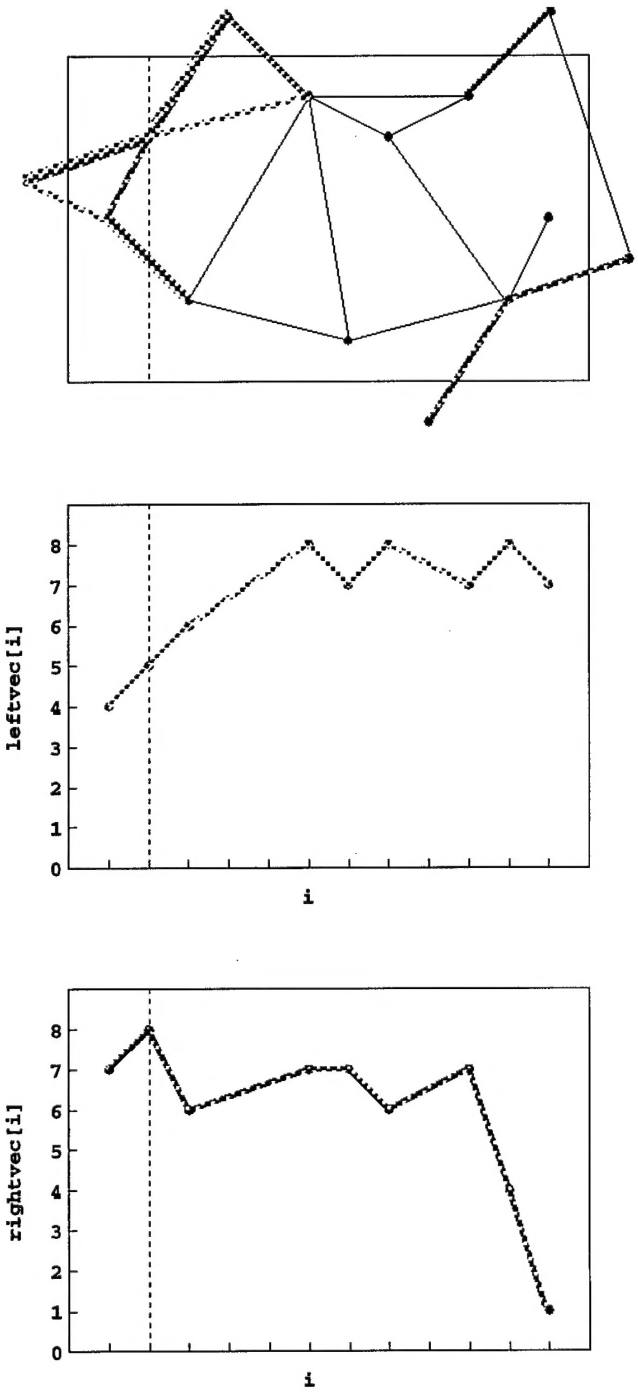


Figure 2: Computation of `leftvec` and `rightvec` in steps 1 and 2 of procedure PARAMETRIC_CUT. The domain to be partitioned (along the x-direction) is given by the top rectangle.

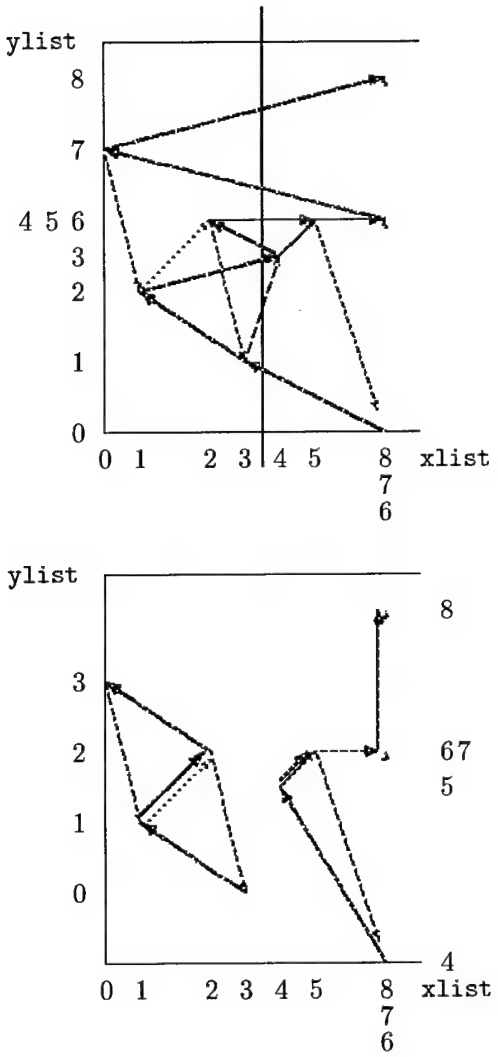


Figure 3: When splitting index lists about the vertical cut, `xlist` (thin red arrows) is split in constant time. `ylist` (thick blue arrows) takes time proportional to the number of points, since each point may have to be moved.

Figure 2 clarifies how `leftvec` and `rightvec` are computed. The vertical dashed lines in this **figure show one possible splitting location,** SPLITPLACE. The value of `leftvec` for this SPLITPLACE is 5. This is because if the right hand region was chosen to be up to and including the node through which this dashed line passes, the number of edges leaving the left hand region (shown in blue) would be 5. Similarly, if the right hand region was chosen to start from this point onwards (i.e., including the node through which the dashed line passes), the number of edges leaving the right hand region (shown in red) would be 8. Note that the edge lying wholly between points outside the region has no impact on the computation. Figure 3 shows how the index lists are split.

Assuming a fixed number of dimensions, the sorts take $O(n \log n)$ time. For point problems, each partition or split takes $O(n)$ time. We therefore get $O(n \log n) + O(dn) = O(n \log n)$ for point problems. For graph problems the sorting time is unchanged. The time to split is now $O(e)$ per level, as we have to look at every edge at every level resulting in $O(\max[n \log n, de])$ time. However in this case it is important to remember that the graphs corresponding to unstructured grids from 2-d aerodynamic problems are planar and thus have $e = O(n)$. Typical 3-d aerodynamic grids have bounded degree and again have $e = O(n)$. Thus we again obtain $O(n \log n)$. The time for fast parametric dissection is thus an improvement over the $O(n \log^2 n)$ time for simple binary dissection, even though parametric dissection uses a more complex partitioning criterion.

5 A Simple Parallel Algorithm

We now discuss a parallel version of the parametric dissection algorithm. This is a simple algorithm that does not utilize the available processors well: its runtime is $O(n)$ *independent* of the number of processors, assuming that the data points are supplied in sorted form. However its extreme simplicity is likely to make its implementation easy and its measured run times may well be competitive with the more complex algorithm presented in Section 6. We start by considering point problems and discuss graph problems (which are only slightly more complicated to implement) at the end of this Section.

We make the reasonable assumption that the partitioning is to be carried out on the same parallel machine on which the problem is to be solved. Thus 2- and 3-d problems are computed on 2- and 3-d meshes, respectively. Alternatively, since a large enough hypercube can have any lower dimensional mesh embedded in it, we may choose to run our problems on hypercubes.

Discussion of a parallel implementation is complicated by the issue of *mapping*. Whereas in the serial algorithm we were only concerned with the partitioning, in the parallel algorithm we would

like to partition our domain and at the same time deliver the resulting subdomains to the correct processors. This can result in substantial savings in time, as discussed below.

The question that now arises is how we are to map the 2^d subdomains that arise after a depth d partitioning onto a $p = 2^d$ processor system. The mapping that we choose is the *natural mapping* described by Berger & Bokhari[1]. When the first bisection is made, dividing the domain into, say, a left half and a right half, then the left subdomain is associated with the left half of the mesh and the right subdomain with the right half. This process is repeated until the subdomains at the dth level are reached—these are associated with individual processors.

5.1 Basic bisection step

Suppose that we have a $p = 2^d$ processor *chain-connected* parallel machine. We shall describe how the basic bisection step is carried out on this chain and then later show how this chain is mapped onto the target parallel machine.[1]

For purposes of illustration, we shall assume that we have a 2-d point problem with n points and that the point data (comprising $<x, y>$ coordinates) has been duplicated and two sorted lists prepared, one for each coordinate. These lists are loaded into our chain in a linear order, with $2n/p$ points per processor.

Sweep-x Sweep through each point of the x-list *sequentially* from left to right, in order to identify the optimal split point. The x-coordinate of the split point is SPLITVALUE.

Migrate-x Move all points of the x-list with x-coordinate $\leq$ SPLITVALUE to the left half of chain and remaining points to the right half.

Mark-y Sweep through the y-list, marking with the label LEFT, those points whose x-coordinates are $\leq$ SPLITVALUE and all others with RIGHT.

Migrate-y Move all points of the y-list marked LEFT (RIGHT) to the left (right) half of the chain.

Each of the above four steps takes time proportional to n. The basic bisection step can now be repeated on the two halves of the chain, with the roles of x and y interchanged. If at each bisection step the number of points is exactly halved, the time required is proportional to

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots < 2n.$$

[1]Which could be a $2^{d/2} \times 2^{d/2}$ 2-d mesh, a $2^{d/3} \times 2^{d/3} \times 2^{d/3}$ 3-d mesh, or a dimension d hypercube.

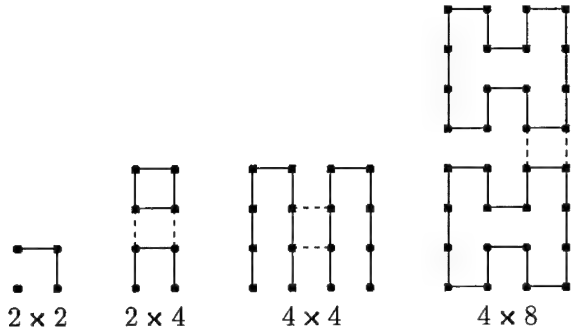


Figure 4: Constructing Bisectionable Chain Embeddings (BCEs).

Parametric binary dissection does not guarantee that each dissection step will exactly halve the number of points. We shall assume that the maximum number of points at every step of the partitioning is a constant times the ideal balance at that step. Thus the $O(n)$ result obtained above holds for simple as well as parametric binary dissection.

5.2 Bisectionable Chain Embedding

The bisection procedure described above only serves to partition the domain over a chain of processors. When carrying out computations on 2-d or 3-d domains we would naturally prefer to use 2- or 3-d meshes for our computation. We now describe embeddings of chains in 2- or 3-d meshes which have the interesting property that when the basic bisection step of Section 5.1 is successively applied to such chains, then the points migrate to the processors on which they should be mapped according to the *natural* mapping. No explicit routing of data blocks is required. This property eliminates an expensive routing step.

Figure 4 shows how a Bisectionable Chain Embedding (BCE) is constructed by combining two smaller BCEs. These curves are similar to Peano's space filling curves. A formal description for the procedure for generating these curves is given in [13]. Figure 5 shows a bisectionable chain embedding of size 16×16. Our sorted x and y-lists are mapped onto this chain starting at • and ending at ■. Application of the basic bisection step (vertical cut) results two sets of sublists, one set starting at • and ending at □; the other starting at ○ and ending at ■. The procedure is repeated with 2 horizontal cuts.

The key property of BCEs is that at this stage the left half of the mesh chain will contain only the points of the original lists that should be mapped onto the left half of the mesh and similarly for the right half of the chain. Thus when the bisection procedure is carried out recursively on a BCE,

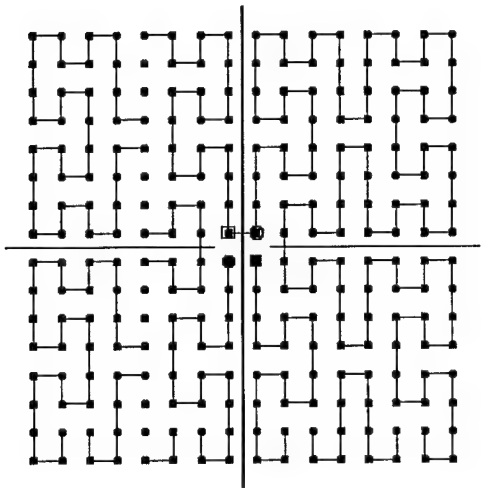


Figure 5: A 16 × 16 BCE. The sorted x and y lists are mapped onto this chain starting at • and ending at ■.

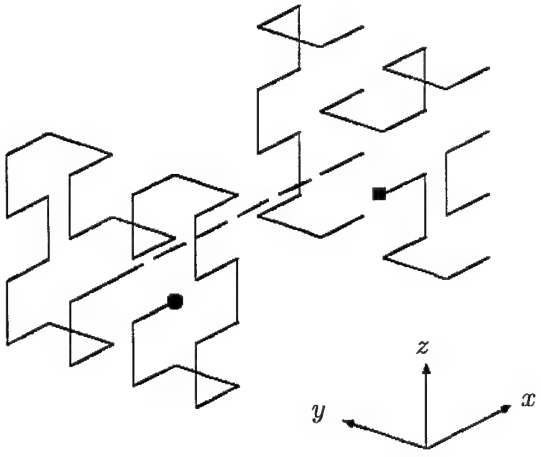


Figure 6: A 3-d BCE of size 4 × 4 × 4. x, y and z lists are mapped onto this chain starting at • and ending at ■ . **The first bisection (with a plane perpendicular to the** x axis) will cut the dashed segment. This is repeated recursively for the y and z directions (spacing along the x axis is distorted).

the data points move to their respective parts of the mesh, so that at the end of the procedure each processor contains its naturally mapped points. The concept of Bisectionable Chain Embeddings is easily extended to higher dimensions (Figure 6).

5.3 Graph Problems

We present our analysis for the case of degree constrained graphs embedded in 3-space, and assume that 3 copies of the graph are available to us, sorted by each of the dimensions[2]. Each item in the x-list, for example, contains the $<x, y, z>$ coordinates of the point and the coordinates of *all* points adjacent to this point. These lists are mapped onto a chain of processors as before and the chain of processors embedded in a 3-d mesh.

The basic bisection step for graph problems requires visiting each processor sequentially, and within each processor, traversing the x-list. As each point is visited, we count the number of edges that would be cut if this point were the extreme point in the bisection. This process is repeated in the reverse direction and then the point where the minimum of $nodes + \lambda \times (edges\ cut)$ occurs is found along the lines of the serial procedure of Section 4. This is followed by list migration. This step is then repeated successively in the y and z directions. Of course, at each point we must visit the nodes adjacent to that point, and list migration involves moving not only each point, but also its adjacent points (i.e., the complete list entry). Our time and space complexity is unchanged at $O(n)$ because we have assumed a constant degree constraint.

6 Fast Parallel Algorithm

The results of the preceding Section suggest an $O(n)$ algorithm for dissection. This is a simple algorithm that does not utilize the available processors well: its runtime is independent of the number of processors available, assuming that the data points are supplied in sorted form. However its extreme simplicity is likely to make its implementation easy and its measured run times may well be competitive with the more complex algorithm presented below.

The ideal algorithm for parametric dissection of an n node problem on a p processor system would have complexity $O(n/p \log n/p)$, which is the same as if each processor were solving the

[2]Applications to higher or lower dimensions are immediate, although it is to be kept in mind that the space required by this algorithm (on each processor) is proportional to the number of dimensions of the problem. It should be recalled that the ultimate objective of the partitioning is to permit a complex aerodynamic computation to take place. The partitioning is carried out *before* the computation. The actual computation requires a large number of variables for each point to store, for example, the velocity vectors, pressure, density etc. Typically from 50 to 100 locations are required for each point [3]. This space can thus freely be used for the binary dissection.

Architecture	Run Time
2-d mesh	$O(\frac{n}{p^{1/2}} + p^{1/2} \log p)$
3-d mesh	$O(\frac{n}{p^{2/3}} + p^{1/3} \log p)$
hypercube	$O(\frac{n}{p} \log^3 p)$

Table 1: Algorithmic complexity of parallel parametric dissection on various architectures.

subproblem resident on it in isolation. This lower bound is difficult to achieve because of the overhead of interprocessor communication. Nevertheless we have succeeded in developing good algorithms for 2 and 3-d meshes and hypercubes. The details of these algorithms are involved and may be found in our earlier technical report [13]. Table 1 summarizes our results.

7 Applications to Unstructured Meshes

A portion of a 2-d unstructured mesh is shown in Figure 7. It can be seen that this mesh has a very large variation in node density. The objective, in generating the mesh, is to have a higher density of nodes in the regions where there is greater need for accuracy. It is this variation in density that makes such meshes difficult to partition. Three-dimensional unstructured meshes are an obvious extension but are difficult to portray on a 2-d page. We show in Figure 8 a 3-dimensional mesh surrounding the wing, fuselage and engine of an aeroplane.

We have implemented the Fast Parametric Dissection algorithm of Section 4, using equation (2) of Section 3. This algorithm has been used to partition several very large 3-d unstructured grids taken from aerodynamic problems. When applying parametric dissection on such grids, it is often the case that the first cut is badly imbalanced as far as the number of nodes is concerned. This is because binary dissection considers the graph to be embedded in a rectangle or cuboid, with edges extending to the sides of the rectangle or cuboid (as shown in Figure 7). The mesh really occupies a roughly ellipsoidal region of 2 or 3-d space (which cannot be depicted in Figure 7 as it is very large compared to the wing cross-section shown). When λ is non zero, the first cut is likely to slice off a small tip of the ellipsoid, so as to minimize the number of edges cut. Thus we have a tiny number of nodes in one region and most of the nodes in the other region. The objective (2) is correctly minimized and the partitioning obtained is superior to a plain partitioning, but only for depth 1. Beyond depth 1 or 2 this poor initial cut leads to bad partitions. This phenomenon is very similar to that described by Stone [10] in connection with the partitioning of random graphs. Our solution to this problem is to carry out the first 1, 2 or 3 cuts with $\lambda = 0$ and switch over to

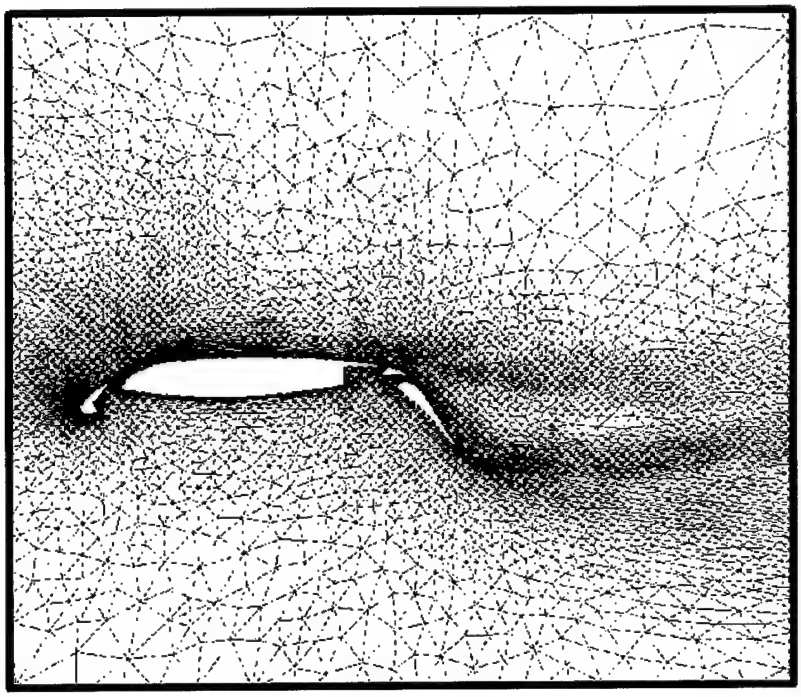


Figure 7: **A 2-d unstructured mesh surrounding the cross section of an aeroplane wing with extended flaps** and slat.

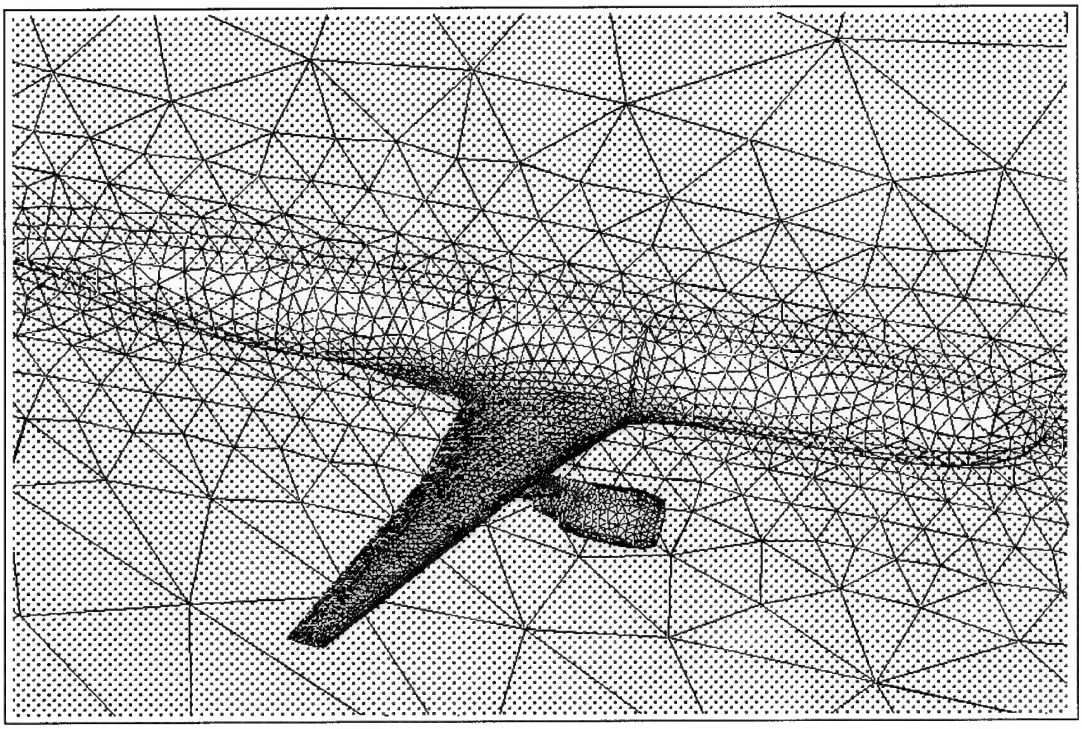


Figure 8: A 3-dimensional unstructured mesh surrounding the wing, fuselage and engine of an aeroplane.

the desired value of λ only after these initial cuts have balanced the number of nodes in the initial 2, 4 or 8 subregions.

In order to evaluate the speedup that would be obtained if a parametric binary dissection were used, compared to plain binary dissection, we carried out an experiment with a 3-d mesh of size 106,064 nodes and 697,992 edges (partly shown in Figure 8). This mesh is derived from a problem involving a wing and engine pod and half a fuselage. Measured run time on a 50 MHz MIPS R4000 processor for a depth 15 partition of this mesh is 83 seconds (excluding time to input the mesh).

The following evaluation procedure was repeated for depths $= 2 - 15$.

- Run the parametric dissection algorithm for various values of λ, starting with 0.
- For each run obtain $maxnodes(\lambda)$ and $maxedges(\lambda)$, the maximum number of edges and nodes over all regions.
- The normalized run time for a dissection is

$$t_{\text{parametric}}(\lambda) = maxnodes(\lambda) + \lambda \times maxedges(\lambda).$$

  This assumes ideal communications on the target parallel processor.

- $maxnodes(0)$ and $maxedges(0)$ are the values that would have obtained if plain binary dissection had been used, since for $\lambda = 0$ parametric dissection reduces to plain dissection. Thus for this problem the time taken by a plain dissection is

$$t_{\text{plain}} = maxnodes(0) + \lambda \times maxedges(0).$$

- For a given value of λ the performance advantage of the parametric algorithm is

$$Improvement(\lambda) = \frac{t_{\text{plain}}}{t_{\text{parametric}}(\lambda)}.$$

The results of the above experiment are summarized in the plots of Figure 9 which show the performance improvement of parametric dissection over plain dissection. Since parametric dissection reduces to plain dissection for $\lambda = 0$, the curve corresponding to this λ is constant at 1.00. There is no improvement for depth=1 and 2 because plain dissection is used for these depths, as discussed above.

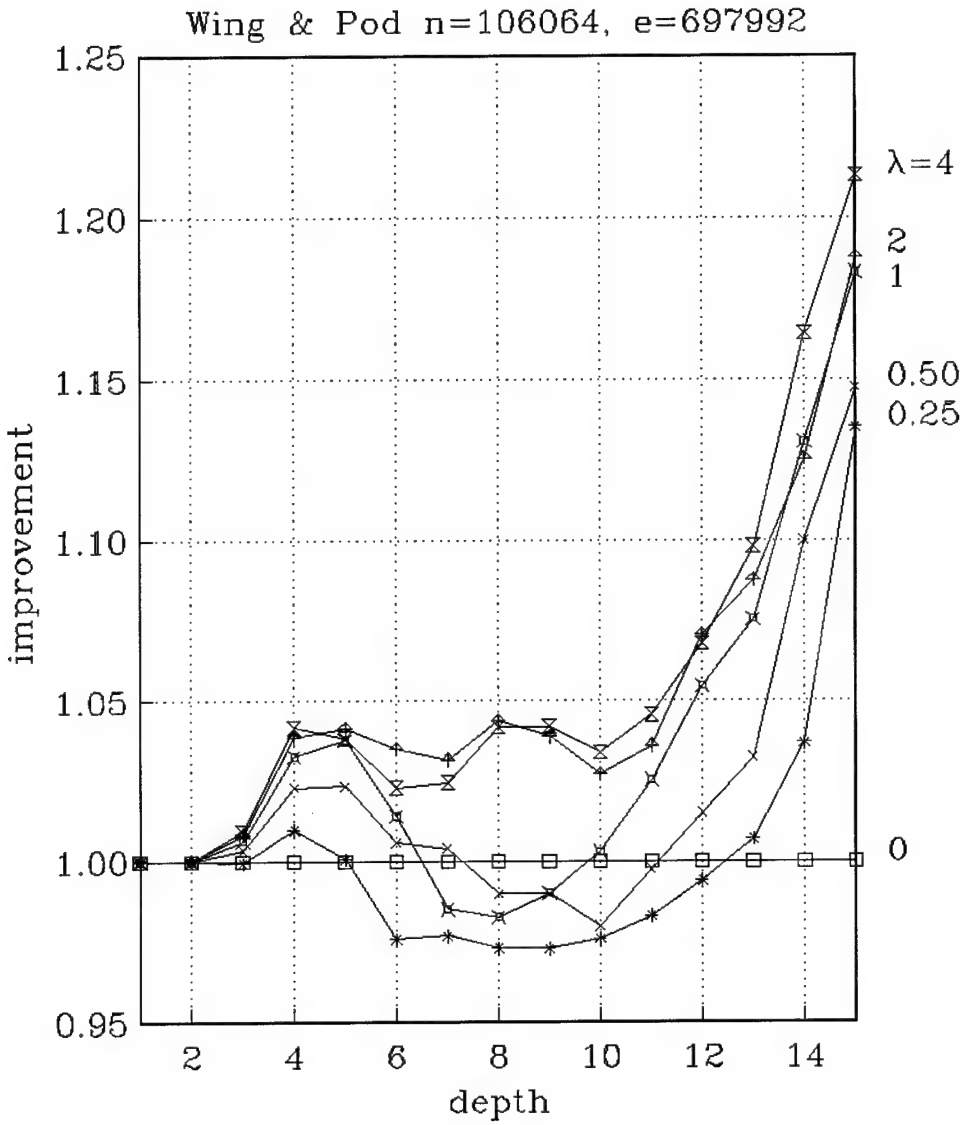


Figure 9: Improvement of parametric binary dissection over plain binary dissection, when applied to a 3-d aerodynamic mesh with $\approx$ 0.1 million nodes and $\approx$ 0.7 million edges, for depths 1-15 (corresponding to $1, 2, 4, \ldots, 32768$ processors). For $\lambda = 0$ parametric dissection reduces to plain binary dissection and there is no speedup.

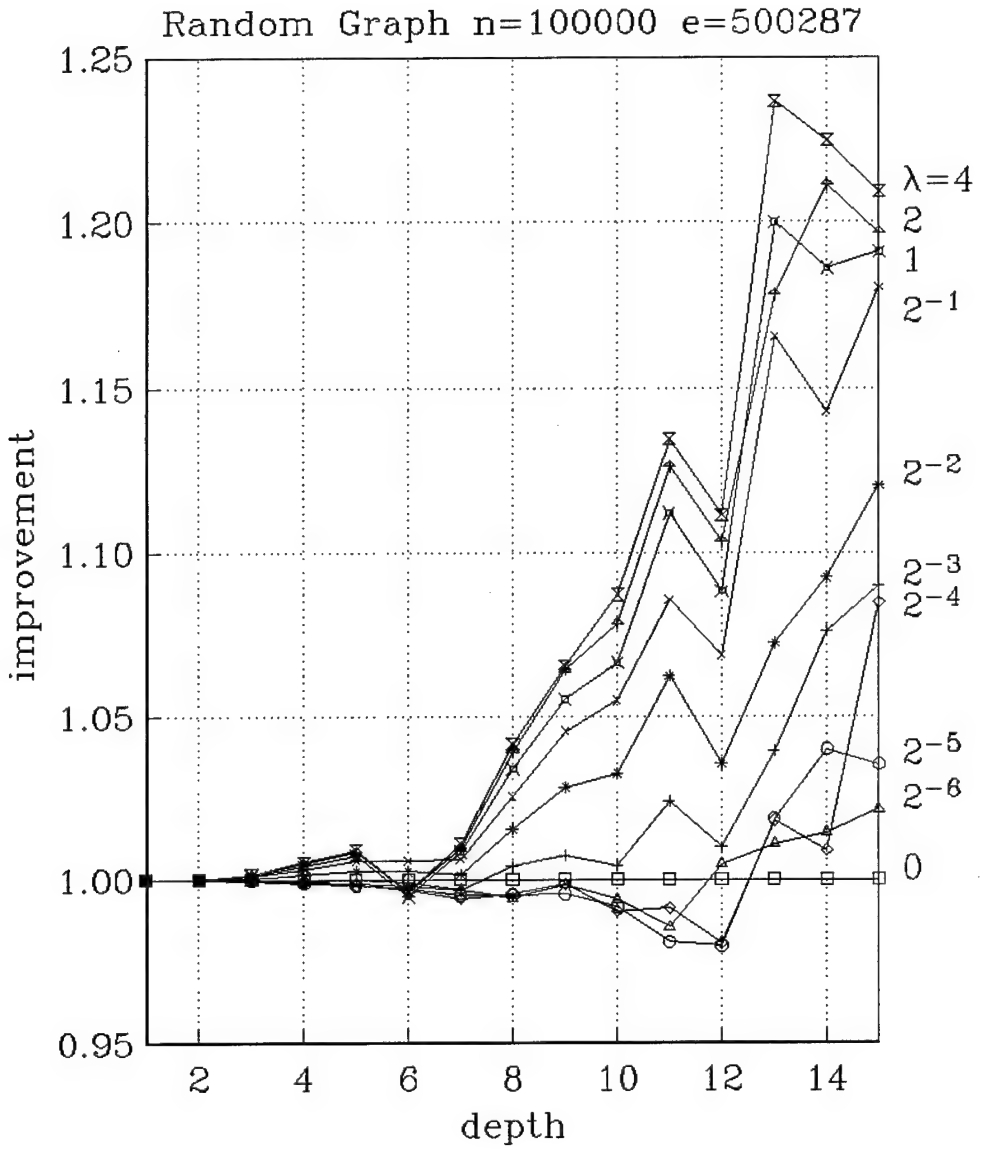


Figure 10: Performance improvement of parametric binary dissection on a random graph with 0.1 million nodes and 0.5 million edges. Performance improvements are obtained over plain binary dissection for all but the smallest values. In practical applications, λ is likely to be greater than 1.

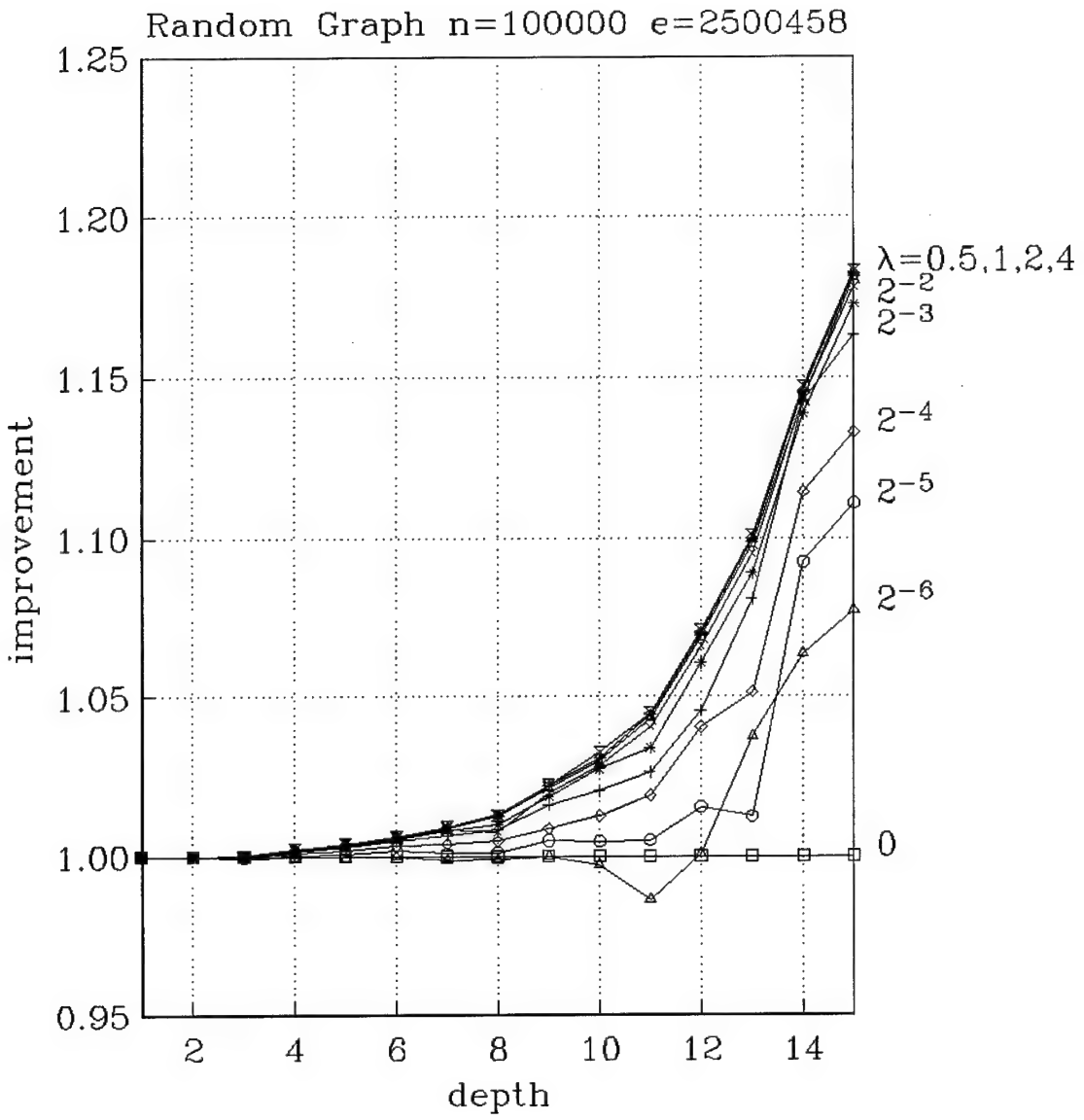


Figure 11: Performance improvement of parametric binary dissection on a random graph with 0.1 million nodes and 2.5 million edges. For this relatively dense graph, performance improvement saturates at about $\lambda = 0.5$.

It can be seen that parametric dissection gives good improvements over plain dissection for $\lambda > 1$ over most of the range of depths. For $\lambda \geq 1$, there is a steep increase in improvement beyond depth 10. For $\lambda = 4$ the improvement is greater than 20% for depth 15. We note that in practical machines the value of λ (the remote/local access time ratio) is usually 1 or greater.

Figures 10 and 11 show the performance of parametric binary dissection on random graphs. Here too the performance improvement is substantial for $\lambda \geq 0.5$.

8 Color Image Quantization with Fast Adaptive Dissection

We now come to the second variant of binary dissection, namely Fast Adaptive Dissection (FAD). This algorithm finds application in *color image quantization*, where samples in a high-resolution color space are mapped onto a lower resolution space in a way that minimizes the color error [7]. More formally, we are given a digital image whose pixels are chosen from a palette containing 2^m colors, and we wish to generate an acceptable reproduction using a palette of 2^k colors, where $k < m$. Typical values for m run from 15–24, while k is usually in the range from 8–12. Color quantization is commonly used to convert full-color images into *colormapped* or *pseudocolor* images in which each pixel is a k-bit index into a color lookup table, or *colormap*. The resulting images are more compact than the originals and are suitable for display using the inexpensive video systems found in most personal computers and many workstations.

In full-color images, the m bits of color information are typically divided into three distinct color components, each using approximately $m/3$ bits. If we assume a red-green-blue (RGB) color model, then each component represents an axis in a three-dimensional color grid. The component values at each pixel can be thought of as indices into this grid. The problem then becomes one of partitioning the grid such that (1) the total number of regions is bounded by 2^k, and (2) the single color value computed for each region serves as a satisfactory representative of all the colors within the region.

8.1 Fast Adaptive Dissection

A variety of heuristic techniques have been proposed for partitioning the color space. Our approach most closely resembles Heckbert's *median cut* algorithm [7], but uses a modified version of Fast Parametric Dissection (Section 4) to speed up operations involving the color samples within regions. These include searching for split points, determination of bounding boxes, and computation of representative colors.

The first step is to scan the original image and record which points in the color space are represented, and how frequently. We store this information in a 3-d histogram matrix (Figure 12).

The histogram is then scanned to produce a list of the colors which occur. The color list is replicated and sorted for each color component, as required by the Fast Dissection algorithm. Since our color components require only a few bits each, we can avoid the level of indirection required by the PARAMETRIC_CUT algorithm of Section 4. Instead, each color component is stored directly as a bit field within a list item, reducing both memory and computation costs.

We next need a strategy for partitioning the lists. In the context of this problem, the two most important criteria are *popularity*, defined as the number of pixels represented by the colors within a region, and *color error*, defined as the maximum distance between points within a region. We have found that a multiplicative relationship between these two parameters produces better images than either of them alone, and is also superior to an additive relationship. Thus our objective function in FAD is *popularity* × *color error*.

In our earlier descriptions of Parametric Binary Dissection and Fast Dissection, we have assumed a recursive partitioning process which descends until the maximum number of subregions is produced, or until a region cannot be subdivided further. For Fast Adaptive Dissection, we follow Heckbert's lead and modify this strategy to utilize *adaptive partitioning*. With adaptive partitioning, the directions of the cuts are not predetermined by the depth of the recursion, but are chosen dynamically based on properties of the data. We discuss this point in more detail below.

Another disadvantage of our original recursive formulation is that the partitioning process can "bottom out" prematurely—one or more branches of the recursion tree may encounter regions which cannot be further subdivided, even though other branches may offer ample opportunity for subdivision. The net result is that some of the available colormap entries go unused. To overcome this problem, we use an iterative variant of Fast Dissection. After each cut is made, the objective function is evaluated for the resulting subregions, and they are placed on a global subregion list, sorted by descending value of the objective function. At each step of the iteration, the first subregion on the list is partitioned. This procedure guarantees that every available colormap entry will be used (assuming the original image contains at least 2^k colors), and gives priority to splitting regions with the largest deviations from the ideal.

Our color quantization algorithm is therefore adaptive in two ways: the direction of the cuts is data-dependent, as is the choice of regions to be split. We call this modified approach *Fast Adaptive Dissection* (or FAD), and refer to color quantization using this technique as *FAD quantization*.

We have found that, for color quantization, the objective function is needed only to determine which region to split next. To find the split point within each region, a simpler and faster heuristic works well: regions are split at the midpoint along their longest edge, i.e., in the direction of largest color error (Heckbert's *median cut* strategy). With the FAD algorithm the midpoint can be found quickly using a binary search on the corresponding sorted sublist. The *popularity* value for each subregion is conveniently tallied during the re-ordering pass on either of the remaining sublists, with the color fields in each list item serving as indices into the original histogram matrix.

To determine *color error*, we use a simple estimate. Rather than searching for the two most extreme points in each region or explicitly computing the error relative to the original image, we use the square of the length of the diagonal of the bounding box for the region. With Fast Dissection, finding the bounding box is trivial—we simply obtain the respective maximum and minimum color components from each of the three sort lists. At each partitioning step, these are directly available via the L and U list indices. A dynamic view of the partitioning process can be found in [14].

When the partitioning phase is complete, the representative color of each region is set to the average of the color values within that region, weighted by the frequency of their occurrence (Figure 13). The collection of representative colors forms the new colormap for the image. All of the histogram entries are then replaced by indices to their representative colors. The sort lists speed up both of these steps, since empty cells in the histogram are not represented in the lists and therefore do not have to be examined.

To complete the process, the original image is re-scanned, and the value of each pixel is replaced with its colormap index, using the histogram matrix as a lookup table.

8.2 Experimental Results

Many color quantization techniques have been developed previously [5][6][7][9][11]; ours is of interest because it produces good results at high speed. Unfortunately, direct comparisons with earlier methods are difficult because

- previous results have been reported over a period of 15 years, spanning several generations of processor technology;
- different methods were tested against different sets of input images; and
- source code from previous implementations is not readily available.

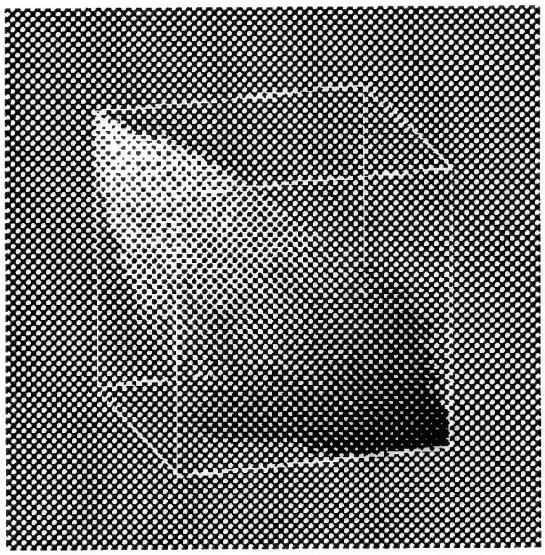


Figure 12: RGB histogram array for the 15-bit image in Figure 14b. Non-zero elements are represented by their corresponding colors.

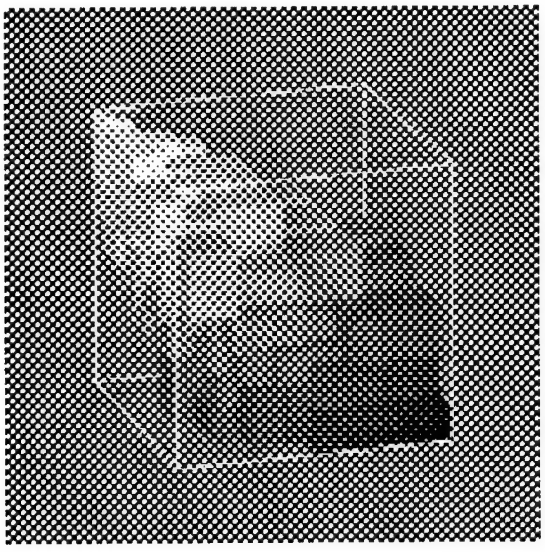


Figure 13: The color space in Figure 12 partitioned into 256 regions using Fast Adaptive Dissection. **The influence of the** *popularity* parameter gives rise to large variations in partition size.

Algorithm Step	Processor Type		
	IBM POWER2 *67 MHz*	SGI/MIPS R4400 *250 MHz*	Sun UltraSPARC *167 MHz*
1. Pre-quantize & histogram	0.1042	0.0836	0.0475
2. Build sort lists	0.0062	0.0098	0.0081
3. Partition color space	0.0130	0.0179	0.0107
4. Build colormap	0.0040	0.0040	0.0031
5. Remap image	0.0344	0.0308	0.0265
Total time	**0.1618**	**0.1461**	**0.0959**

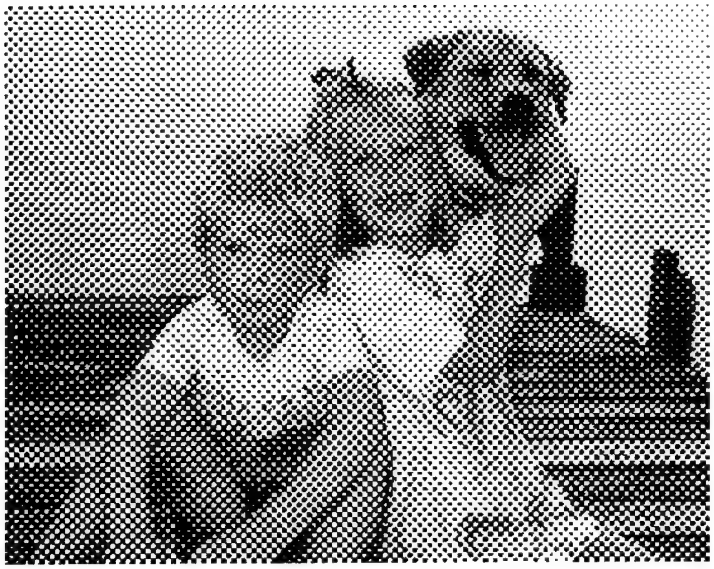
Table 2: Runtimes (in seconds) for FAD quantization.

However, we believe that our method is competitive with existing techniques. While higher image quality may be obtained with more computational effort, and faster results can be achieved at lower image quality, the FAD algorithm provides a good balance for applications in which both speed and quality are important.

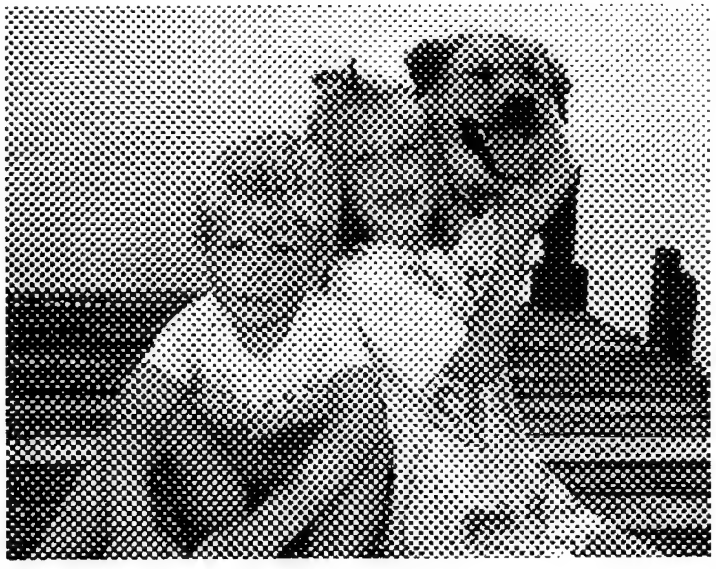
We have implemented FAD quantization in the C language and tested it on several workstation platforms. Table 2 shows execution times for the test image shown in Figure 14a. As part of the histogramming step, we reduce the color precision of the original image from 24 to 15 bits. This simplification, employed by many color quantization algorithms, decreases memory and processing requirements significantly while degrading the original image only slightly (Figure 14b).

As the timings indicate, the pixel-level operations on the image data (steps 1 and 5) dominate the execution time, comprising from 78–85% of the total. The partitioner (steps 2 and 3) requires 12–19% of the total time, while computation of the representative colors and construction of the mapping table (step 4) requires a miniscule 3%.

Although the fast dissection method partitions the color space rapidly (step 3), we must take care that the overhead for the initial sorts (step 2) doesn't outweigh the benefits. We have found that the fastest way to sort the lists is to simply scan the histogram array in the desired order, appending non-zero entries to the list as we encounter them. This is easily accomplished by using a triply-nested loop (one for each dimension) with the appropriate sorting index (R, G, or B) at the outermost level. For example, the following C code fragment produces a list of colors sorted

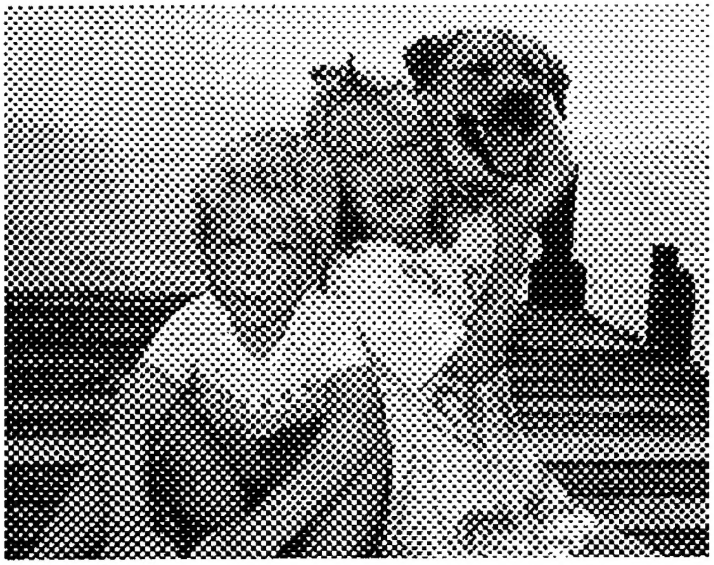


(a) Original 24-bit image (141813 colors).

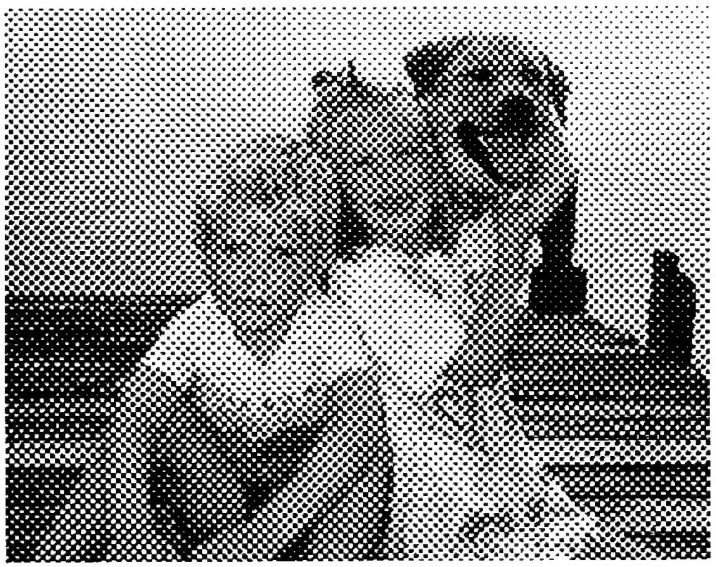


(b) Uniform quantization to 15 bits (6980 colors). QRMSE = 4.2.

Figure 14: Test image (559 × 436 pixels).



(c) Uniform quantization to 8 bits. QRMSE = 22.0. Only 140 colors are used; the remaining 116 lie in regions of the color space which are not represented in the input image. Note the severe color banding in the flesh tones and background, and loss of highlight detail in the t-shirts.



(d) FAD quantization to 256 colors. QRMSE = 9.2. Some color banding persists, but details re-emerge in the background and on the shirts.

Figure 14 *(cont'd)*: Test image (559 × 436 pixels).

along the G axis:

```
const int DIM = 32, MAXCOLORS = DIM * DIM * DIM;
unsigned histogram[DIM][DIM][DIM];
int r, g, b, i;
unsigned rlist[MAXCOLORS], glist[MAXCOLORS], blist[MAXCOLORS];
                ...
for (i = 0, g = 0; g < DIM; g++)
   for (b = 0; b < DIM; b++)
      for (r = 0; r < DIM; r++)
         if (histogram[r][g][b])
            glist[i++] = (r << 10) | (g << 5) | b;
```

A similar loop is used to sort in the B direction. For the R sort, we can optimize further by taking advantage of C's row-major storage order for arrays:

```
unsigned *h;
int j;
                ...
h = (unsigned *) histogram;
for (i = j = 0; i < MAXCOLORS; i++)
   if (h[i])
      rlist[j++] = i;
```

Our tests indicate that this sorting strategy is an order of magnitude faster than a tuned quicksort, and is therefore essential in realizing the performance gains of the FAD approach.

Future work in this area shall develop along the following lines:

1. Improvements in the parallel PBD algorithm. Communication overhead shows up prominently in the expressions for run time of our algorithm. Whether this can be reduced significantly is an open question.

2. Implementations of the parallel versions of the dissection algorithms for mesh architectures such as the Intel Paragon and ASCI Red machines.

3. Evaluation of the performance of PBD on a large set of unstructured meshes.

4. Use of these dissections for actual computation, especially for aerodynamic problems.

5. Applications of PBD/FAD to other areas, such as partitioning problems in circuit and VLSI design.

Acknowledgments

We are grateful to Dimitri Mavriplis for many useful discussions. We thank Clyde Gumbert for providing us with a large 3-d mesh for experimentation. The test image in Section 8 was supplied by Susan Crockett and is used with permission. Craig Gotsman graciously provided additional images for comparison testing. We are grateful to M. Y. Hussaini, M. Salas and K. E. Durrani for their encouragement of this research.

References

[1] Marsha J. Berger and Shahid H. Bokhari, A partitioning strategy for non-uniform problems across multiprocessors. *IEEE Transactions on Computers*, C-36:570–580, May 1987.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[3] D. J. Mavriplis, Three dimensional unstructured multigrid for the Euler equations. *AIAA Journal* 30(7):1753–1761, July 1992.

[4] Karen M. Dragon and John L. Gustafson, A low-cost hypercube load-balance algorithm. In *The Fifth Conference on Hypercubes, Concurrent Computers and Applications*, Mar. 1989, pp. 583–589.

[5] J. L. Furlani, L. McMillan, and L. Westover. Adaptive colormap selection algorithm for motion sequences. *Proceedings ACM Multimedia '94*, Oct. 1994, pp. 341–347.

[6] Michael Gervautz and Werner Purgathofer. A simple method for color quantization: octree quantization. In *Graphics Gems*, A. Glassner, ed., Academic Press, 1990, pp. 287–293, .

[7] P. Heckbert, Color image quantization for frame buffer display. *Computer Graphics*, 16(3):297–307, July 1982.

[8] A. Pothen, H. D. Simon and K. P. Liou, Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Mathematical Analysis and Applications*, 11:430–452, 1990.

[9] E. Roytman and C. Gotsman, Dynamic color quantization of video sequences. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):274–286, Sept. 1995.

[10] Harold S. Stone, *High Performance Computer Architecture.* Addison-Wesley, Reading, Massachusetts, 1990, pp. 311-313.

[11] **Xiaolin Wu, Efficient statistical computations for optimal color quantization.** In *Graphics Gems II*, J. Arvo, ed., Academic Press, 1991, pp. 126–133.

[12] F. P. Preparata and M. I. Shamos, *Computational Geometry: An introduction.* Springer-Verlag, 1985.

[13] S. H. Bokhari, T. W. Crockett and D. M. Nicol, Parametric Binary Dissection. ICASE Report No. 93-39, NASA Contractor Report 191496, July 1993. Available at `ftp://ftp.icase.edu/pub/techreports/93/93-39.ps.Z`.

[14] T. W. Crockett, S. H. Bokhari and D. M. Nicol, "Color Image Quantization using Fast Adaptive Dissection," `http://www.icase.edu/reports/supplements/97/97-29.FAD.html` (MPEG animation), June 1997.